CSE 421

Dynamic Programming

RNA, Sequence Alignment

Yin Tat Lee

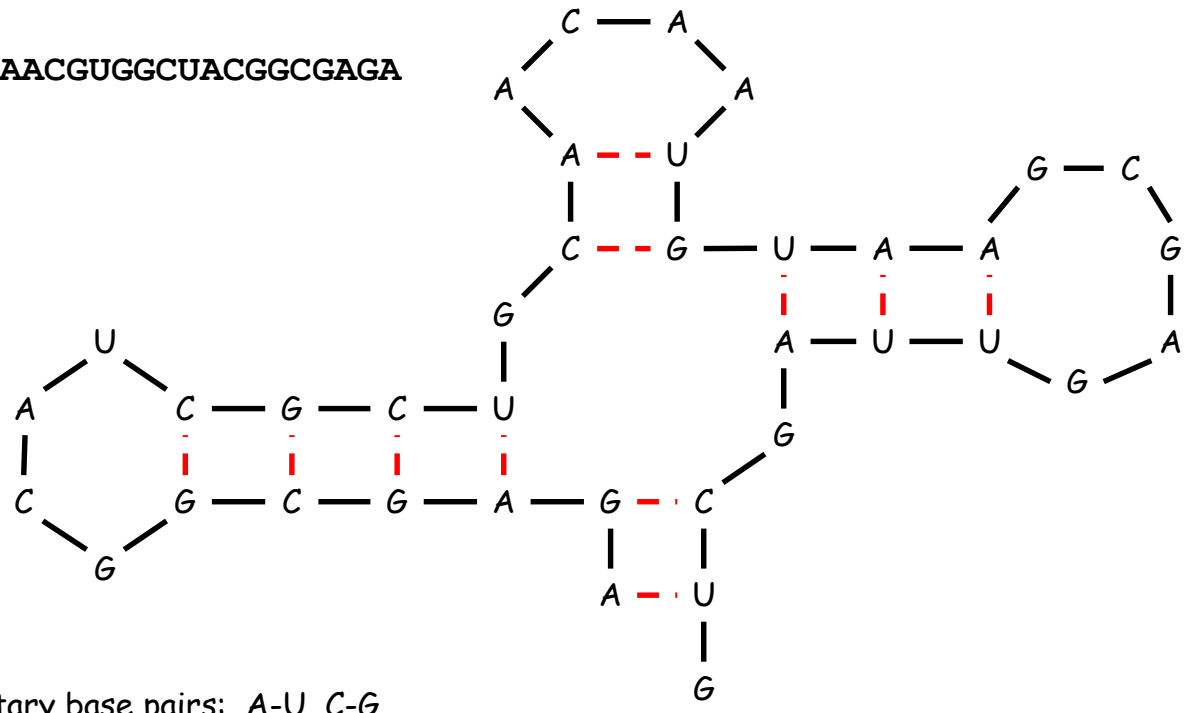
RNA Secondary Structure

RNA Secondary Structure

RNA: A String B = $b_1 b_2 \ldots b_n$ over alphabet { A, C, G, U }.

Secondary structure. RNA is single-stranded so it tends to loop back and form base pairs with itself. This structure is essential for understanding behavior of molecule.

Ex: **GUCGAUUGAGCGAAUGUAACAACGUGGCUACGGCGAGA**



complementary base pairs: A-U, C-G

RNA Secondary Structure (Formal)

Secondary structure. A set of pairs $S = \{(b_i, b_j)\}$ that satisfy:

[Watson-Crick.]

- S is a *matching* and
- each pair in S is a Watson-Crick pair: $A - U$, $U - A$, $C - G$, or $G - C$.

[No sharp turns.]: The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.

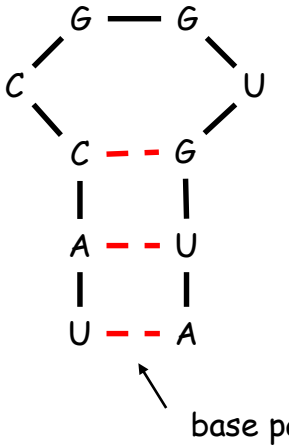
[Non-crossing.] If (b_i, b_j) and (b_k, b_l) are two pairs in S, then we cannot have $i < k < j < l$.

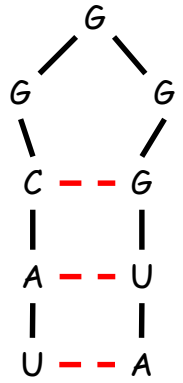
Free energy: Usual hypothesis is that an RNA molecule will maximize total free energy.

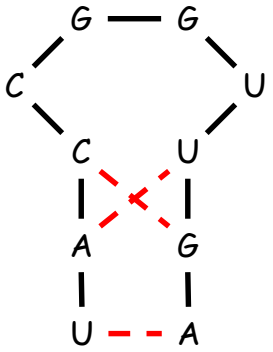
approximate by number of base pairs

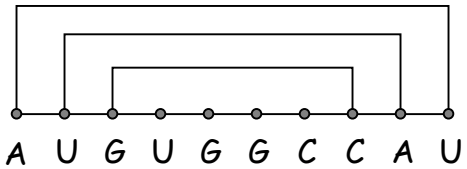
Goal: Given an RNA molecule B = $b_1 b_2 \ldots b_n$, find a secondary structure S that maximizes the number of base pairs.

Secondary Structure (Examples)

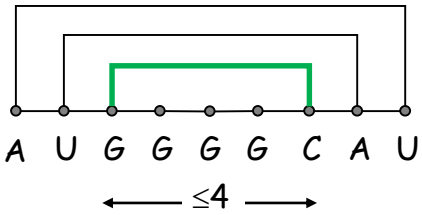




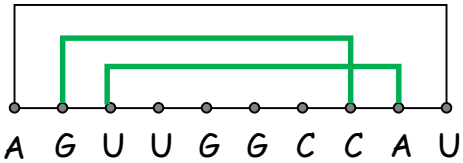




ok



sharp turn



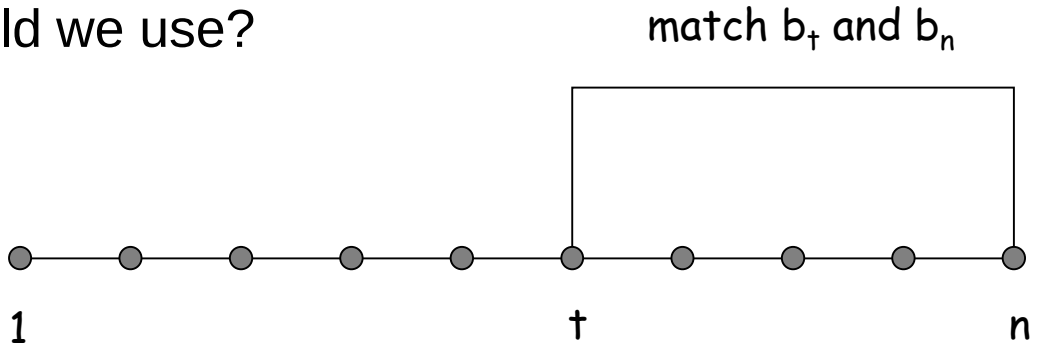
crossing

DP: First Attempt

First attempt. Let $OPT(n)$ = maximum number of base pairs in a secondary structure of the substring $b_1 b_2 \ldots b_n$.

Suppose b_n is matched with b_t in $OPT(n)$.

What IH should we use?



Difficulty: This naturally reduces to two subproblems

- Finding secondary structure in $b_1, \ldots, b_{t-1}$, i.e., OPT(t-1)
- Finding secondary structure in $b_{t+1}, \ldots, b_{n-1}$, ???

DP: Second Attempt

Definition: $OPT(i,j)$ = maximum number of base pairs in a secondary structure of the substring $b_i, b_{i+1}, \dots, b_j$

The most important part of a correct DP; It fixes IH

Case 1: If $j - i \leq 4$.

- $OPT(i,j) = 0$ by no-sharp turns condition.

Case 2: Base b_j is not involved in a pair.

- $OPT(i,j) = OPT(i,j-1)$

Case 3: Base b_j pairs with b_t for some $i \leq t < j - 4$

- non-crossing constraint decouples resulting sub-problems
- $OPT(i,j) = \max_{i \leq t < j-4} \{ 1 + OPT(i,t-1) + OPT(t+1,j-1) \}$

Recursive Code

```
Let M[i,j]=empty for all i,j.

Compute-OPT(i,j){
   if (j-i <= 4)
     return 0;
   if (M[i,j] is empty)
      M[i,j]=Compute-OPT(i,j-1)
      for t=i to j-5 do
         if (b_t,b_j is in {A-U, U-A, C-G, G-C})
           M[i,j]=max(M[i,j], 1+Compute-OPT(i,t-1) +
                      Compute-OPT(t+1,j-1))
   return M[j]
}
```

Does this code terminate?
What are we inducting on?
Key question: is there any loop in the recursion?

Formal Induction

Let $OPT(i,j)$ = maximum number of base pairs in a secondary structure of the substring $b_i, b_{i+1}, \dots, b_j$

Base Case: $OPT(i,j) = 0$ for all i,j where $|j-i| \leq 4$.

IH: For some $\ell \geq 4$, Suppose we have computed $OPT(i,j)$ for all i,j where $|i-j| \leq \ell$.

IS: Goal: We find $OPT(i,j)$ for all i,j where $|i-j| = \ell+1$. Fix i,j such that $|i-j| = \ell+1$.

Case 1: Base b_j is not involved in a pair.

- $OPT(i,j) = OPT(i,j-1)$ [this we know by IH since $|i-(j-1)| = \ell$]

Case 2: Base b_j pairs with b_t for some $i \leq t < j-4$

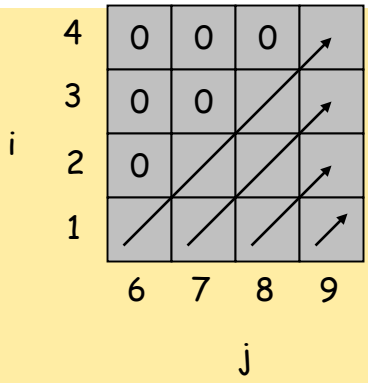
- $OPT(i,j) = \max_{i \leq t < j-4} \{ 1 + OPT(i,t-1) + OPT(t+1,j-1) \}$

We know by IH since difference $\leq \ell$

Bottom-up DP

```
for ℓ = 1, 2, …, n-1
  for i = 1, 2, …, n-1
    j = i + ℓ
    if (ℓ <= 4)
      M[i,j]=0;
      else
        M[i,j]=M[i,j-1]
        for t=i to j-5 do
          if (b_t, b_j is in {A-U, U-A, C-G, G-C})
            M[i,j]=max(M[i,j], 1+ M[i,t-1] + M[t+1,j-1])

  return M[1, n]
}
```



Running Time: $O(n^3)$

Lesson

We may not always induct on i or w to get to smaller subproblems.

We may have to induct on $|i - j|$ or $i + j$ when we are dealing with more complex problems.

Sequence Alignment
(Edit distance)

Word Alignment

How similar are two strings?

ocurrance

occurrence

o	c	u	r	r	a	n	c	e	-
o	c	c	u	r	r	e	n	c	e

5 mismatches, 1 gap

Quiz

Input: two n-bit strings s_1 and s_2.

- $s_1 = AGGCTACC$
- $s_2 = CAGGCTAC$

Output: minimum number of insertions/deletions to transform s_1 into s_2.

Algorithm: ????

Even if the objective is precisely defined, we are often not ready to start coding right away!

After this quarter, you should able to solve it by $O(n^2)$ time.

Open problem: Is $O(n^{1.999})$ possible? (Probably not due to Strong ETH)

Message: Don't be discouraged if it takes 30 years to solve a problem.

However, limit your time of each homework to just 10 years. There are 8 homework.

2

Best paper on FOCS 2018

Approximating Edit Distance Within Constant Factor in Truly Sub-Quadratic Time*

Diptarka Chakraborty[1], Debarati Das[2], Elazar Goldenberg[3], Michal Koucký[4,*], and Michael Saks[5]

[1,2,4]Computer Science Institute of Charles University, Malostranské náměstí 25, 118 00 Praha 1, Czech Republic
[3]The Academic College Of Tel Aviv-Yaffo, School of Computer Science, Tel Aviv-Yaffo, Israel
[5]Department of Mathematics, Rutgers University, Piscataway, NJ, USA

July 13, 2018

See arXiv:1804.04178

o	c	-	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

1 mismatch, 1 gap

o	c	-	u	r	r	-	a	n	c	e
o	c	c	u	r	r	e	-	n	c	e

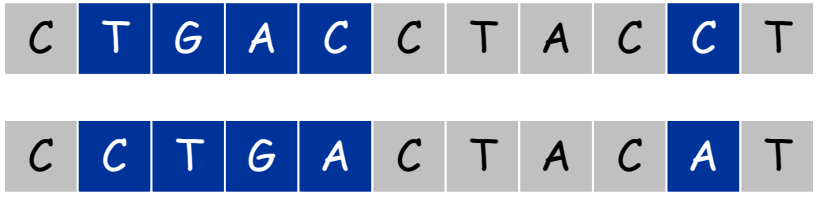
0 mismatches, 3 gaps

Edit Distance

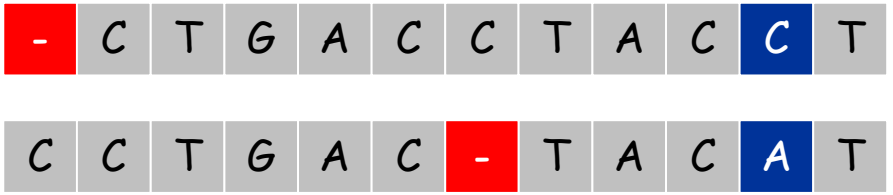
Edit distance. [Levenshtein 1966, Needleman-Wunsch 1970]
Cost = # of gaps + #mismatches.

Applications.

- Basis for Unix diff and Word correct in editors.
- Speech recognition.
- Computational biology.



Cost: 5



Cost: 3

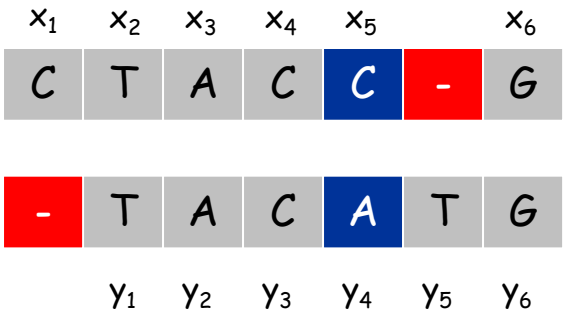
Sequence Alignment

Given two strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ find an alignment with minimum number of mismatch and gaps.

An alignment is a set of ordered pairs $(x_{i_1}, y_{j_1}), (x_{i_2}, y_{j_2}), \ldots$ such that $i_1 < i_2 < \cdots$ and $j_1 < j_2 < \cdots$

Example: `CTACCG` vs. `TACATG`.
Sol: We aligned
x_2-y_1, x_3-y_2, x_4-y_3, x_5-y_4, x_6-y_6.

So, the cost is 3.



DP for Sequence Alignment

Let $OPT(i, j)$ be min cost of aligning $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$

Case 1: OPT matches x_i, y_j

- Then, pay mis-match cost if $x_i \neq y_j$ + min cost of aligning $x_1, \ldots, x_{i-1}$ and $y_1, \ldots, y_{j-1}$ i.e., $OPT(i-1, j-1)$

Case 2: OPT leaves x_i unmatched

- Then, pay gap cost for x_i + $OPT(i-1, j)$

Case 3: OPT leaves y_j unmatched

- Then, pay gap cost for y_j + $OPT(i, j-1)$

```
M[i, j] = min( (x_i=y_j ? 0:1) + M[i-1, j-1],
               1 + M[i-1, j],
               1 + M[i, j-1])
```

Induction

What is the order of induction? (i.e. why there is no loop?)
We can do induction on $i + j$.

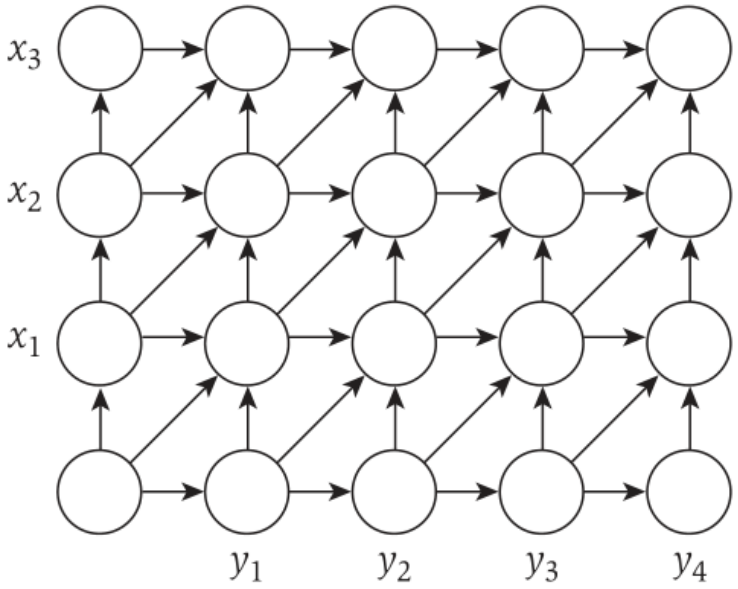


Figure 6.17 A graph-based picture of sequence alignment.

Bottom-up DP

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn) {
  for i = 0 to m
     M[0, i] = i
  for j = 0 to n
     M[j, 0] = j

  for i = 1 to m
     for j = 1 to n
        M[i, j] = min( (xi=yj ? 0:1) + M[i-1, j-1],
                       1 + M[i-1, j],
                       1 + M[i, j-1])
  return M[m, n]
}
```

Analysis: $\Theta(mn)$ time and space.

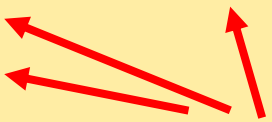
Computational biology: m = n = 100,000. 10 billions ops OK, but 10GB array?

Optimizing Memory

If we are not using strong induction in the DP, we just need to use the last (row) of computed values.

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn) {
  for i = 0 to m
     M[0, i] = i
  for j = 0 to n
     M[j, 0] = j

  for i = 1 to m
     for j = 1 to n
        M[i, j] = min( (xi=yj ? 0:1) + M[i-1, j-1],
                       1 + M[i-1, j],
                       1 + M[i, j-1])
  return M[m, n]
}
```



Just need $i-1, i$ rows to compute M[i,j]

DP with $O(m+n)$ memory

- Keep an Old array containing values of the last row
- Fill out the new values in a New array
- Copy new to old at the end of the loop

```
Sequence-Alignment(m, n, x1x2...xm, y1y2...yn) {
  for i = 0 to m
     O[i] = i
  for i = 1 to m
     N[0]=i
     for j = 1 to n
        N[j] = min( (xi=yj ? 0:1) + O[j-1],
                    1 + O[j],
                    1 + N[j-1])
     for j = 1 to n
        O[j]=N[j]
  return N[n]
}
```

M[i-1, j-1]

M[i-1, j]

M[i, j-1]

```
M[i, j] = min( (x_i=y_j ? 0:1) + M[i-1, j-1],
               1 + M[i-1, j],
               1 + M[i, j-1])
```

Shortest Path

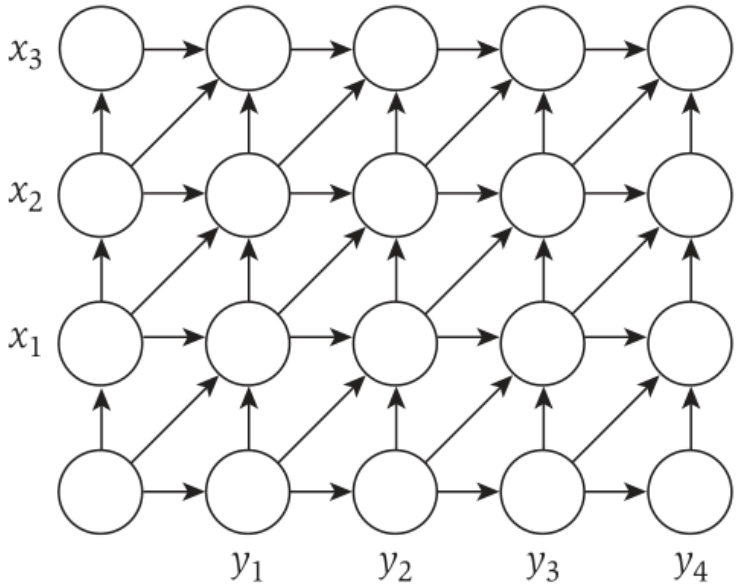


Figure 6.17 A graph-based picture of sequence alignment.

How to recover the alignment?

Hint: bidirectional search.

Lesson

Advantage of a bottom-up DP:

It is much easier to optimize the space.

By the way, edit distance

- can be computed in $O(\frac{n^2}{\log^2 n})$ (1980).
- can be approximated by log factor in $O(n^{1+\varepsilon})$ (~2010).
- cannot be solved in $O(n^{2-\delta})$ exactly (2015).
- can be approximated by O(1) factor in $O(n^{1+\varepsilon})$ (~2018).

Longest Path in a DAG

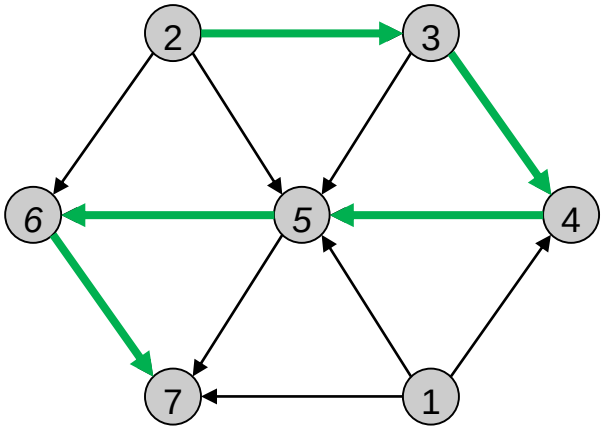
Longest Path in a DAG

Goal: Given a DAG G, find the longest path.

Recall: A directed graph G is a DAG if it has no cycle.

This problem is NP-hard for general directed graphs:

- It has the Hamiltonian Path as a special case



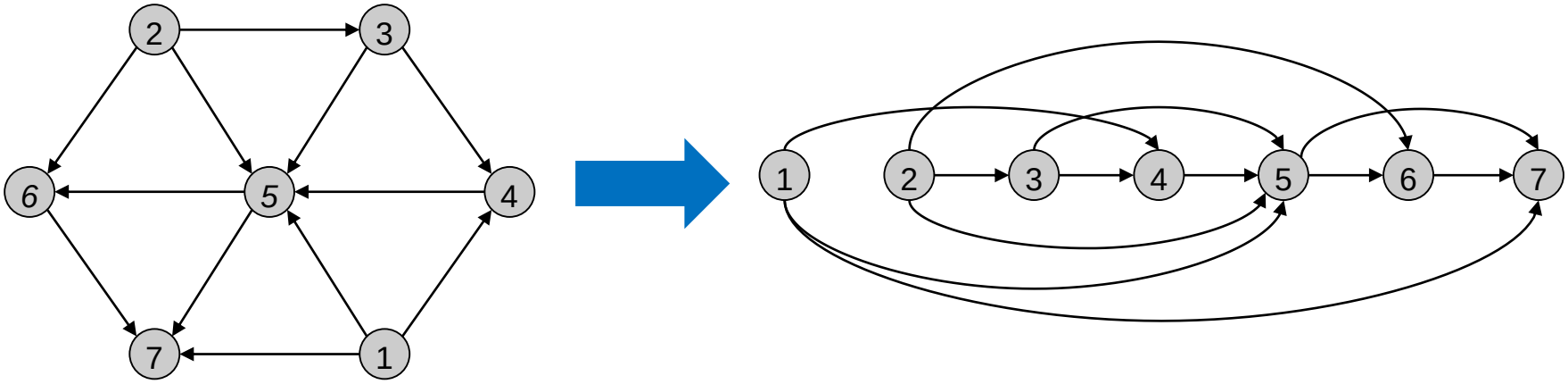
DP for Longest Path in a DAG

Q: What is the right ordering?

Remember, we have to use that G is a DAG, ideally in defining the ordering

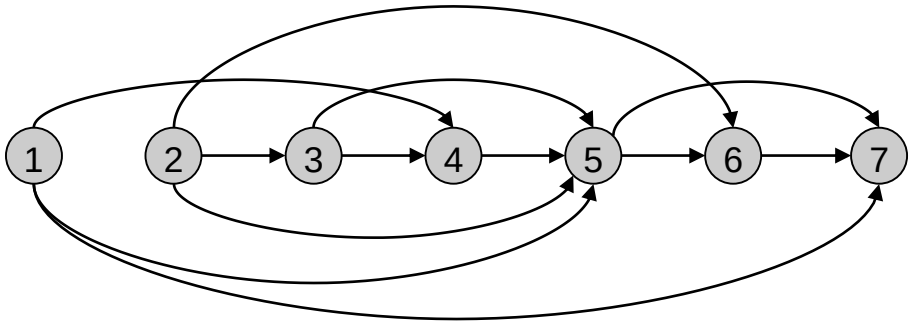
We saw that every DAG has a topological sorting

So, let's use that as an ordering.



DP for Longest Path in a DAG

Suppose we have labelled the vertices such that (i, j) is a directed edge only if $i < j$.



Let $OPT(j)$ = length of the longest path ending at j

Suppose OPT(j) is $(i_1, i_2), (i_2, i_3), \ldots, (i_{k-1}, i_k), (i_k, j)$, then

Obs 1: $i_1 \leq i_2 \leq \cdots \leq i_k \leq j$.

Obs 2: $(i_1, i_2), (i_2, i_3), \ldots, (i_{k-1}, i_k)$ is the longest path ending at i_k.

$$OPT(j) = 1 + OPT(i_k).$$

DP for Longest Path in a DAG

Suppose we have labelled the vertices such that (i, j) is a directed edge only if $i < j$.

Let $OPT(j)$ = length of the longest path ending at j

$$OPT(j) = \begin{cases} 0 & \text{If } j \text{ is a source} \\ 1 + \max_{i:(i,j) \text{ an edge}} OPT(i) & \text{o.w.} \end{cases}$$

Outputting the Longest Path

```
Let G be a DAG given with a topological sorting: For all edges
(i,j) we have i<j.
Initialize Parent[j]=-1 for all j.
Compute-OPT(j){
   if (in-degree(j)==0)
     return 0
   if (M[j]==empty)
     M[j]=0;
     for all edges (i,j)
       if (M[j] < 1+Compute-OPT(i))
         M[j]=1+Compute-OPT(i)
         Parent[j]=i
   return M[j]
}
Let M[k] be the maximum of M[1],…,M[n]
While (Parent[k]!=-1)
   Print k
   k=Parent[k]
```

Record the entry that we used to compute OPT(j)